

Java Entwickler / Software Developer (m/w/d)



Bücher eröffnen Welten – wir tun es ihnen gleich. Als Deutschlands größter Sortimentsbuchhändler verknüpfen wir echte Kundennähe mit richtungsweisender Digitalinnovation. Wir denken das Buch als Erlebnis – ob vor Ort, online, als App oder über den e-Reader. Und versammeln rund 4000 Expertinnen und Experten mit einer Mission: Ein Kulturgut auf allen Kanälen erfahrbar zu machen.

Für unsere Zentrale in **Münster** suchen wir **ab sofort** einen

Java Entwickler / Software Developer (m/w/d)

Welche Aufgaben erwarten dich:

- Du betreibst die eigenverantwortliche Weiterentwicklung unserer **eCommerce-Plattform** mit Spaß an Software-Engineering und Clean Code in einem agilen Produktteam
- Du betreust als **Full-Stack-Entwickler** die ganze Kette vom Datenbankzugriff/ Serviceaufruf bis zur Gestaltung und Auslieferung des Frontends

- Du sicherst die Qualität durch **Code Reviews**, Unit-/Integrations-/End-To-End-Tests sowie die Auslieferung über **Jenkins** und **Git**
- Du misst anhand von **Kennzahlen** die Produktivität, Performance und Qualität eurer Software, um sie stetig zu verbessern
- Du entwickelst zusammen mit dem Product Owner die beste Lösung für unsere **Kunden**

Was bringst du mit:

- Mehrjährige Berufserfahrung in der Software-Entwicklung mit Java, gerne im Bereich **eCommerce**
- Sehr gute Kenntnisse aktueller Frameworks und Konzepte (z.B. **Spring** MVC/Spring Boot, JPA, Hibernate, REST API, Microservices, Self-contained Systems) und Entwicklungswerkzeuge (Git, Maven, Jenkins)
- Know How im Umgang mit dem **Apache Tomcat**, Apache Webserver, Linux, Docker und Co
- Erfahrungen mit **Datenbanksystemen** (MySQL, Postgres) und **Frontend-Technologien** (HTML5, JavaScript, CSS3)
- DDD, TDD, BDD - die Abkürzungen sind Dir nicht unbekannt und idealerweise hast Du hier bereits praktische Erfahrung gesammelt
- Konzeptionelle Stärke, analytisches Denken, Kommunikationsfähigkeit und **sehr gute Deutschkenntnisse**

Diese Benefits bieten wir dir:

- Ein **motiviertes und selbstorganisiertes Produktteam**, dem ein tolles Miteinander und Freude an der Arbeit wichtig ist
- Ein modernes Arbeitsumfeld und die Möglichkeit neue Technologien oder Vorgehensmodelle einzuführen
- Ein **aktueller Technologie Stack** (Java, Apache Tomcat, Spring, Spring Boot, JPA, Maven, Git, Jenkins, Nexus, Gitlab, Sonar, Jira, Eclipse, IntelliJ, Postgres, MySQL, RabbitMQ, Apache Kafka, Handlebars, Grafana, Graylog)
- Besuch und Mitgestaltung von **Fachkonferenzen, Hackathons** und die Möglichkeit auf dem **Thalia Tech Blog** zu veröffentlichen

- Ein familienfreundliches Unternehmen mit **flexiblen Arbeitszeiten** und 30 Tagen Urlaub im Jahr sowie eine Vielzahl an Sozialleistungen

Haben wir dein Interesse geweckt?

Dann bewirb dich unter Angabe deines Gehaltsrahmens und deines frühestmöglichen Eintrittstermins über unser [Online-Portal](#) oder über bewerbungen@thalia.de!

Thalia Bücher GmbH | Deine Ansprechpartnerin: Stefanie Klein

Artikeldaten in Datenbanken: Von einem relationalen Schema zu „Dokumenten“ (2/2)

Teil 2 - Technische Alternativen und deren Vergleich

Die im [ersten Teil](#) beschriebene Problemstellung („... (wir) führen ... die Artikeldaten aus unterschiedlichen Quellsystemen zusammen. Jedes Quellsystem pflegt seine eigenen Update-Frequenzen und -Formen.“) wurde in unserer Systemlandschaft vor einigen Jahren bereits in einer relativ leicht zu beschreibenden aber technisch herausfordernden Anwendung umgesetzt - weiter unten als „Legacy Variante“ beschrieben.

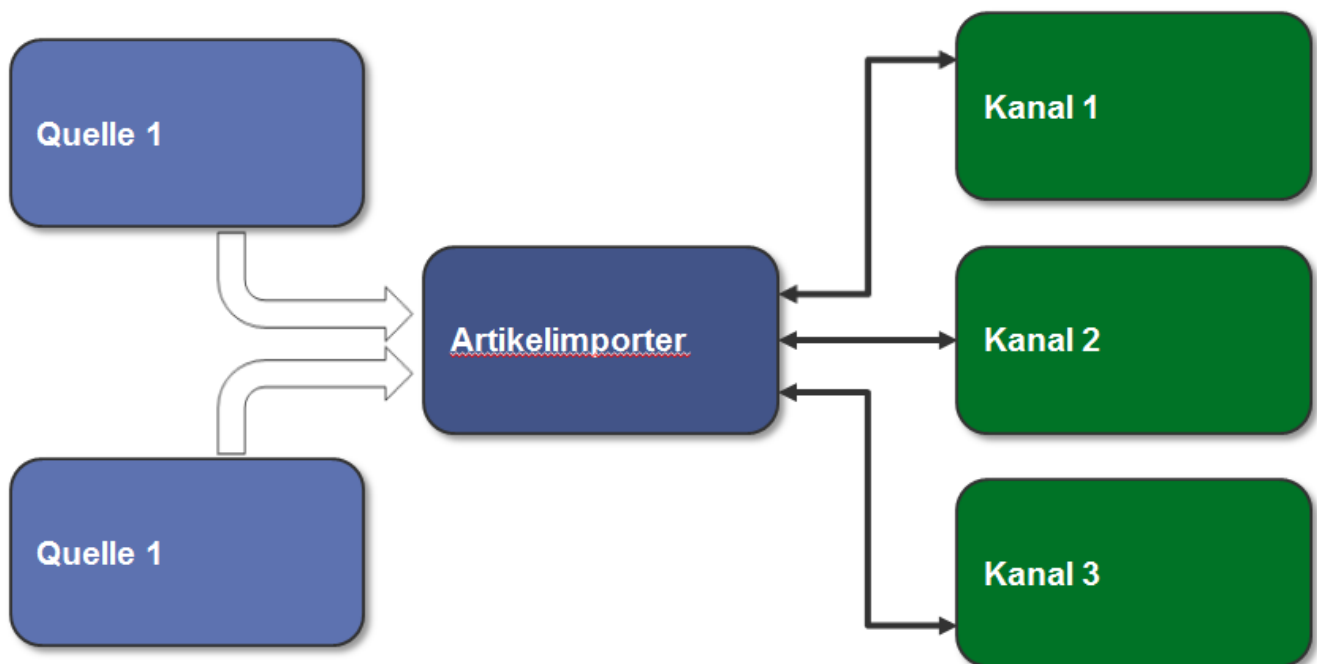
Fachliche Designfehler machten vor kurzem einen Neuentwurf der Anwendung [notwendig](#). Diesen haben wir zum Anlass genommen, u.a. das Thema „Dokumente“ und deren Technologie zu erproben und so auch für andere, zukünftige Anwendungsfälle Erfahrung zu sammeln. Unsere ersten Messungen und unser Vorgehen bei der Auswahl der passenden Technologie soll Inhalt dieses

Teils sein.

Die Beispielanwendung / das Szenario

Besagte Anwendung, im weiteren Text der Einfachheit halber „Artikelimporter“ genannt und etwas simplifiziert, erhält aus zwei Quellen Datensätze. Die Kombination aus einem Datensatz mit seinem Zwilling aus der anderen Quelle ergibt einen Artikeldatensatz. Das bedeutet, die Anwendung muss

1. Die Daten aus den beiden Quellen „verheiraten“, d.h. Datensatz-Hälften zu einem vollständigen Satz zusammenstellen (auf Basis eines einheitlichen Schlüssels).
2. Den kompletten Datensatz eines Artikels den Online-Kanälen (z.B. eReader, Webshop, App usw.) zur Verfügung stellen.
3. Neue „Paare“, Änderungen an diesen Datensätzen oder komplette Rückzüge (bei „Verlust“ eines Zwillings) propagieren.



Diese Anwendung haben wir (mit ein paar anderen Konzepten) neu gebaut und daraufhin den Datenstrom aus unserer Produktion abgegriffen und durch die neuen Prototypen geleitet.

Die unten dokumentierten Messungen sind also nicht aus einem schnuckeligen synthetischen Labortest gekommen („wir füllen die ... mit einer Millionen ... und machen dann ... und wiederholen das dann ... mal“), sondern auf Basis echter

Anforderungen erhoben und mit der bestehenden Legacy Variante verglichen worden.

Bisheriger Ansatz - Legacy

In der ersten Version haben wir einen generativen Ansatz zur Erstellung des DB Schemas auf Basis der Struktur der empfangenen Nachrichten (aus beiden Quellen) gewählt ([HyperJaxb](#)).

So konnten wir gerade in der Anfangszeit, als die Struktur der Datensätze noch volatil war, einiges an Implementierungsarbeit sparen. Die Persistenz wurde mit [JPA](#) realisiert. Folgerichtig ist das Datenschema stark normalisiert (jenseits 30 Tabellen für den beschriebenen Anwendungsfall).

Das Schreiben in das Datenbankschema findet generell in mehreren, parallel arbeitenden Threads statt. Dabei verteilen wir die Datensätze per Modulo Funktion auf der Artikelnummer, um die korrekte Reihenfolge der Bearbeitung innerhalb eines einzelnen Artikels zu sichern. Als Framework bot sich hier Spring Integration an.

Einige Nachteile dieses Entwurfs wie Fehleranfälligkeit und Code-Komplexität sollen nicht Teil dieses Textes sein, wohl aber die bislang erreichte Performance:

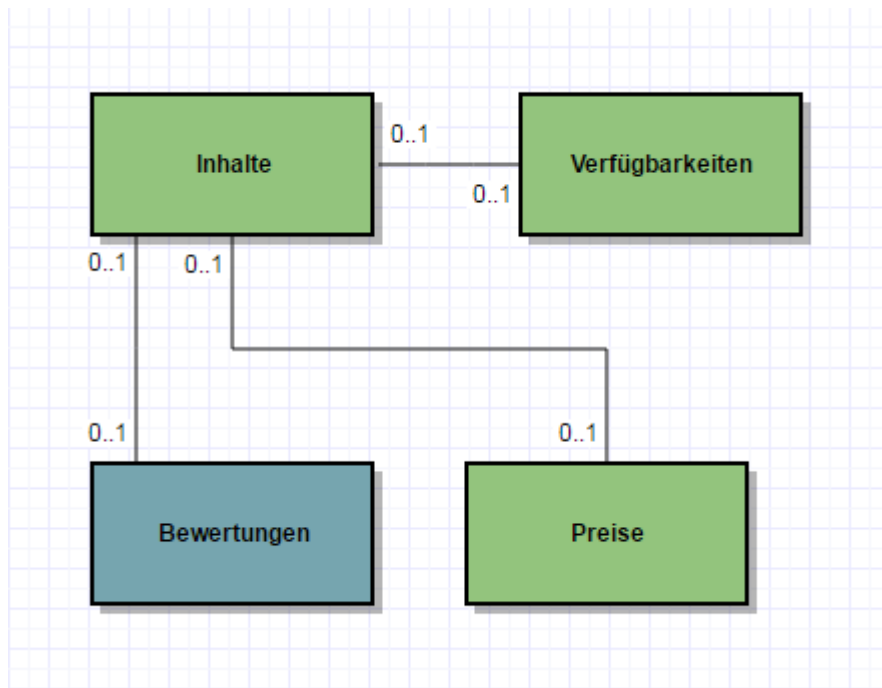
Die PIM Datensätze werden relativ konstant an die Anwendung geliefert und überschreiten im Normalfall nicht die 40k/Stunde. Auf der WWS Seite ist die Anlieferung eher stoßweise, die dann natürlich möglichst schnell übernommen werden müssen. Hier kommen dann auch mal 200k auf einmal.

Neuer Ansatz - PoC

Im ersten Teil des Artikels wurde schon einiges über den Hintergrund und die Designentscheidung bezüglich der Methodik beim Abspeichern berichtet. Hier eine kurze Zusammenfassung:

- Anstatt eine maximale/saubere Normalisierung der Daten anzustreben, wollen wir auf Speicherung von „Dokumenten“ setzen. Gründe dafür sind u.a. die gesteigerte Flexibilität bei Erweiterungen und weniger Performance/Deadlock Themen auf DB Ebene durch radikale Reduktion der Komplexität des Schemas.
- Wir führen ein Dokument pro Datenquelle (in Produktion sind es

inzwischen sechs verschiedene Datenquellen) ein, um Konflikte zwischen den Quellen beim Schreiben zu vermeiden und nicht unnötig viele Daten ändern zu müssen, wenn eine Aktualisierung ansteht.



Hier sei betont, dass in diesem Schaubild Dokumente und nicht etwa Objekte oder Klassen eingezeichnet sind. Das hat Auswirkung auf die Kardinalitäten und die Benennung, z.B. handelt es sich bei „Preise“ um null bis **ein** Dokument, in dem ein oder **mehrere** Preise abgelegt sind.

Die neue Anwendung basiert also auf wenigen Dokumenten statt vieler Klassen. Die DB Statements sind mit Spring-JDBC implementiert.

Auswahl des DBMS / Vergleich der Alternativen

Wir haben im Folgenden das bestehende System mit der alternativen Implementierung auf unterschiedlichen DBMS verglichen.

„Dokumente“ && „JSON“ == NoSQL

Der Umgang mit Dokumenten drängt zur Beschäftigung mit dokumentenbasierten Datenbanken aus dem Topf der [„NoSQL“-Lösungen](#).

Bei dem Schlagwort „NoSQL“ und „Dokumente“ kommt an einer Betrachtung von [MongoDB](#) fast nicht vorbei. Abgesehen davon, dass MongoDB von allen NoSQL/Dokumente-Lösungen unbestritten das beste Marketing hat, ist für uns

der „Generalisten“-Ansatz interessant („Allzweckdatenbank“).

Wir können und wollen uns keinen Zoo von Technologien halten, in dem jede Speziallösung ein paar Prozente besser ist als ein eventueller Generalist – unter der Voraussetzung, dass sie denn optimal konfiguriert/betrieben wird, wozu bei der Anzahl von Speziallösungen keine vernünftig große Gruppe von Menschen in der Lage ist.

Worauf ich hinaus will: wenn wir einen Spezialisten einführen, dann muss er sehr deutlichen Mehrwert mitbringen, der den Aufwand der Einführung und des Betriebs rechtfertigt. Er sollte kein „one-trick pony“ und auch in zwei Jahren noch von Mitgliedern des Produktteams betreibbar sein. In NoSQL einzusteigen mit einer Lösung, die sich selbst als Generalist vermarktet und daher verspricht, auch die Lösung für zukünftige Probleme zu sein, ist da sehr sexy.

Auf der MongoDB Seite findet man sich auch bestätigt, was die obigen Überlegungen angeht, wird hier doch auch eine [„EAV“-Diskussion für Artikeldaten](#) geführt.

Stark, damit haben wir unseren ersten Kandidaten!

„Dokumente“ && „JSON“ != NoSQL

Aber muss es denn was komplett neues sein? Seien wir ehrlich, die Aussicht auf was ganz neues – den „heißen Scheiß“ – macht Spaß. Wir alle wollen ausprobieren, wovon das Internet und die [Regenbogenpresse](#) der Entwickler spricht.

Wir müssen die Systeme allerdings auch betreiben – länger als sein Jahr, in Produktion, mit dramatischen Umsatzausfällen bei Ausfall verbunden. Der Kater droht also immer – wir sind da [stark sensibilisiert](#).

Eine kleine Quellenforschung bei den zwei freien DBMS, die in unserem Haus im Einsatz sind, zeigt, dass man auch dort auf JSON/Dokumente aufgesprungen ist. Also haben wir sie in den Vergleich aufgenommen.

Postgres && JSON

Postgres unterstützt den JSON-Datentyp seit der Version 9.2 und erweitert zur Verfügung stehende Funktionen seitdem. Mit Version 9.4 ist [JSONB](#)

hinzugekommen, der für einige Anwendungsfälle effizienter arbeitet, Indizierung ermöglicht und für unseren Vergleich auch verwendet wurde. Die JSON(B)-Funktionen sowie die gesamte Notation zum Traversieren der Dokumente sind Postgres-spezifisch. (Eine schnelle Einführung gibt es [hier](#).)

Skeptiker könnten jetzt sagen, dass sei nur ein verkleideter „TEXT“-Datentyp mit ein paar Funktionen drum herum, womit sie natürlich größtenteils recht haben. Im hier beschriebenen Anwendungsfall reicht aber genau dieser Grad der Integration aus, um unsere Anforderungen umzusetzen(!). Solange wir jeweils komplette Dokumente in der DB Tabelle austauschen (was wir aufgrund unseres oben beschriebenen Schnitts der Dokumente tun) und nicht Einzelfelder innerhalb der Dokumente editieren, sind wir mit diesem DBMS im Geschäft.

Die Indizierung auf Einzelfelder innerhalb der Dokumente ist möglich.

MySQL & JSON

Siehe da, MySQL unterstützt ebenfalls einen [JSON Datentyp](#) ab Version 5.7. Hier gelten gleiche Aussagen wie bei der Postgres-Variante. Wir haben einen neuen Datentyp, der für die Definition einer DB Spalte verwendet werden kann. Gegenüber der Verwendung eines einfachen String-Feldes verspricht die Datenbank eine Validierung auf syntaktische Korrektheit und ein optimiertes Speicherformat für schnellen Lookup von Einzelfeldern.

Die Indizierung auf Einzelfelder innerhalb eines Dokumentes ist nicht möglich – dafür gibt es aber einen kleinen [Kniff](#), der für unseren Anwendungsfall ausreicht.

Performance Vergleich: MongoDB, Postgres, MySQL

Vier verschiedene Implementierungen haben wir gegeneinander antreten lassen.

Das haben auch schon andere vor uns gemacht. Die gesetzten Bedingungen, der jeweilige Anwendungsfall, die Intention der Autoren, die Validität der Ergebnisse bei älteren Versionsständen u.v.m. lässt den Skeptiker des Öfteren zweifeln, zumal die Ergebnisse sich zum Teil widersprechen. ([Beispiel für einen eventuellen Interessenskonflikt](#))

Für uns war es wichtig, den Vergleich innerhalb unserer Umgebung mit unseren

Versionsständen und unserem Hintergrundwissen durchzuführen – zumal der Aufwand sich aufgrund der freien Verfügbarkeit der DBMS und der vorhandenen umfangreichen Dokumentation in Grenzen hält. Niemand sollte allerdings die gemessenen Werte hernehmen und anfangen, einstellige prozentuale Abweichungen zwischen den Alternativen auszurechnen. Hier geht es darum, ein Gefühl für die richtige Entscheidung zu entwickeln – von gewichteten Bewertungsmatrizen sehe ich daher ab.

In der **ersten Ecke** steht der ursprüngliche Artikelimporter: generiertes DB-Schema, normalisiert, JPA als Persistenz-Framework, Abfragen gegen MySQL Datenbank in Version 5.6. Aus Bequemlichkeit steht dieser Anwendung ein größerer Server zur Verfügung als den Alternativen – aber nichts besonders Bemerkenswertes, siehe Tabelle unten.

In der **zweiten Ecke** steht die neue Version des Artikelimporters, bei uns intern auch als „Nitro-Edition“ bezeichnet (um mal Erwartungsmanagement zu betreiben...), basierend auf zwei Quellen für Dokumente (WWS und PIM), Spring-JDBC, **Postgres in Version 9.5**.

In der **dritten Ecke** läuft die Nitro-Edition gegen **MySQL 5.7** – mit entsprechenden Syntaxanpassungen.

Ecke Nummer vier, bereit sie alle in den Schatten zu stellen, ist eine Nitro-Edition auf Basis **MongoDB in Version 3.3**. Hier musste natürlich mehr angepasst werden als zwischen den beiden gestandenen RDBMS. Das nicht vorhandene Transaktionshandling muss im Code berücksichtigt werden – total hemdsärmelig darf man hier nicht vorgehen.

	Artikelimporter (MySQL 5.6)	Nitro Edition (PostgreSQL 9.5)	Nitro Edition (MySQL 5.7 (uc))	Nitro Edition (MySQL 5.7 (c))	Nitro Edition (MongoDB 3.3)
Anzahl DB Tabellen	30+	2	2	2	1 (Collection)
Dauer WWS Meldung	486	3	7	3	1
Dauer 7 PIM Meldungen	127	26	41	28	23
Lines of Code	Ca. 16k	Ca. 1600	-	-	-
Hardware	8 Kerne, 8 GB RAM	4 Kerne, 4 GB RAM	4 Kerne, 4 GB RAM	4 Kerne, 4 GB RAM	4 Kerne, 4 GB RAM

Der Vergleich MySQL zu Postgres war zunächst verblüffend. Auffällig war vor

allem der benötigte Platz auf dem Laufwerk für die gespeicherten Daten. Ein Studium der MySQL Seiten offenbarte, dass die Tabellen tunlichst mit dem Parameter „row_format=compressed“ erstellt werden sollten und als Voraussetzung dafür der Server mit dem „innodb_file_format“ „Barracuda“ (statt „Antelope“) laufen muss. Das Resultat war eine Reduktion des benötigten Speicherplatzes um ca. 50% (und damit auf Postgres Niveau) und ebenfalls fast eine Halbierung der benötigten Ausführungszeiten (!). Guckst du [hier](#) für die offizielle Dokumentation. In der oben abgebildeten Tabelle haben wir zwei MySQL Zeiten aufgenommen – mit (c) und ohne (uc) Komprimierung.

Entscheidung für Technologie

Nach der kleineren Nachbesserung auf MySQL Seite kann man auf Basis der Tabelle folgende objektive Schlüsse ziehen:

- Der neue Entwurf der Anwendung ist dem alten weit überlegen in allen Faktoren. Gute Entscheidung. Gut gelernt.
- Postgres und MySQL tun sich nicht viel im Bereich Ressourcen und Performance. Auf keinen Fall sind Größenordnungen dazwischen.
- MongoDB hat die höchste Geschwindigkeit erzielt.

Schaut man also nur auf die reine Geschwindigkeit, ist MongoDB den beiden RDBMS davongezogen – je nach Betrachtungsweise um ein paar Prozent bis Faktoren. In Relation zur bisherigen Lösung aber sind alle Umsetzungsvarianten schnell – schneller als sie sein müssten, denn PIM und WWS können die Nachrichten gar nicht so schnell generieren wie sie hier abgearbeitet werden (das wäre auch peinlich).

Die größten angepriesenen [Stärken](#) von MongoDB sind für uns allerdings nicht relevant:

- Skalierung über Rechenzentren hinweg bis der Geldbeutel aufgibt.
- Ausfallsicherheit bis der Geldbeutel aufgibt.
- Flexibelste Indizierung für „mächtige Abfragen“.
- Textsuche

In unserem konkreten Szenario haben wir von der überlegenen Leistung von MongoDB also nichts, dafür aber ein paar Nachteile.

- Wir kennen uns mit der Benutzung nicht aus (z.B. fehlende Transaktionen).
- Wir kennen uns mit dem Betrieb nicht aus (Früherkennung von Zipperlein, Sizing).
- Wir haben ungeklärte Support-Szenarien (Standzeit, Wer kennt sich aus?).
- Kosten für die Enterprise Lizenz – speziell wenn man vom Einzelserver abrückt.

Ich habe anfangs geschrieben, die Lösung muss „sehr deutlichen Mehrwert mitbringen, der den Aufwand der Einführung und des Betriebs rechtfertigt“. Diesen Mehrwert sehen wir hier nicht.

Die beiden anderen DBMS tun sich in den Messwerten nicht viel. Hier kommen eher pragmatische Argumente:

1. Die Postgres Syntax für Abfragen auf Teildokumenten ist aus meiner Sicht einfacher zu verstehen. Unsere Anwendung lädt allerdings nur ganze Dokumente, so dass diese Funktionalität nicht zum Tragen kommt.
2. Die Indizierung auf Feldern innerhalb von Dokumenten ist in Postgres vorgesehen. In der MySQL Variante muss man [über virtuelle Spalten](#) gehen, was zwar funktioniert, aber Applaus gibt es dafür von mir nicht.
3. Für Abwärtskompatibilität benutzen wir Datenbank Views, die dem abfragenden ein normalisiertes Schema vorgaukeln. Dazu müssen wir Arrays innerhalb von Dokumenten in DB Zeilen ausmultiplizieren. Beispielsweise müssen alle Preise aus dem Dokument „Preis“ in jeweils einer Zeile angezeigt werden. Das ist unter Postgres eine schicke Funktion („[jsonb_array_elements](#)“), aber unter MySQL muss man ein [join mit einer Hilfstabelle](#) machen – geht, aber ebenfalls kein Applaus von mir. Hier verspricht MySQL 8 ein wenig Abhilfe, ich würde bis dahin aber nicht die Luft anhalten wollen.
4. Wir haben hier bei Thalia eine große MySQL Installation zur Auslieferung von Artikeldaten in Betrieb. Das beinhaltet sowohl die Ausfallsicherheit, die Skalierung/Lastverteilung als auch die korrekte Netzwerkzone. Unsere Postgres-Installation ist zwar vom Volumen her größer, steht aber am falschen Ende der Firma ☐

Am Ende der Entscheidungsfindung war der letzte, hoch pragmatische Punkt

ausschlaggebend. Hier haben wir auf „time-to-market“ geachtet in dem Wissen, dass wir bei so einfachen Queries die Systeme Postgres/MySQL ohne nennenswerten Aufwand jederzeit wechseln können.

Also – Überraschung! – wir haben uns für **MySQL** entschieden!